PRIDE: Practical Intrusion Detection in Resource Constrained Wireless Mesh Networks

Amin Hassanzadeh, Zhaoyan Xu, Radu Stoleru, Guofei Gu, and Michalis Polychronakis*

Department of Computer Science and Engineering, Texas A&M University, USA
* Computer Science Department, Columbia University, USA
{amin, z0x0427, stoleru, guofei}@cse.tamu.edu, mikepo@cs.columbia.edu

Abstract. As interest in wireless mesh networks grows, security challenges, e.g., intrusion detection, become of paramount importance. Traditional solutions for intrusion detection assign full IDS responsibilities to a few selected nodes. Recent results, however, have shown that a mesh router cannot reliably perform full IDS functions because of limited resources (i.e., processing power and memory). Cooperative IDS solutions, targeting resource constrained wireless networks impose high communication overhead and detection latency. To address these challenges, we propose PRIDE (PRactical Intrusion DEtection in resource constrained wireless mesh networks), a non-cooperative real-time intrusion detection scheme that optimally distributes IDS functions to nodes along traffic paths, such that detection rate is maximized, while resource consumption is below a given threshold. We formulate the optimal IDS function distribution as an integer linear program and propose algorithms for solving it accurately and fast (i.e., practical). We evaluate the performance of our proposed solution in a real-world, department-wide, mesh network.

Keywords: wireless mesh network, intrusion detection, resource constraints, integer linear programming, real-world implementation.

1 Introduction

Wireless Mesh Networks (WMN) are self-managing networks that provide Internet, intranet, and other services to mobile and fixed clients using a multi-hop multi-path wireless infrastructure consisting of mesh nodes [1, 2]. They have emerged as a cost-effective broadband network technology for services in large remote areas where no networking infrastructure is available, e.g., rural connectivity in Zambia [3] and disaster response applications [4]. A wireless mesh network can serve as the backbone communication infrastructure among WiFi networks, ad hoc networks, sensor networks and the Internet [4]. It is important to remark the lack of a vantage point for the network traffic, due to the peer-to-peer nature of communication in WMN.

As the interest in WMN grows, security issues, especially intrusion detection, become of paramount importance. Due to the *decentralized nature of WMN*, researchers have proposed distributed solutions for network wide intrusion detection. Distributed solutions do not rely on a single vantage point (e.g., gateways in traditional intrusion detection systems (IDS) in wired networks) as there always could be internal traffic (e.g., between two hosts[1]) in WMN to be monitored. The state-of-the-art distributed solutions can be categorized as: i) *monitoring node* solutions; and ii) *cooperative* solutions. *Monitoring node* solutions [5,6] assign the same set of IDS functions (i.e., detection rules) to monitoring nodes (note: each monitoring node is responsible for a distinct part of the network). These solutions, however, have high false negative rates. This is because some IDS functions cannot be executed on monitoring nodes with limited resources (e.g., processing power and memory). A recent work [7] investigates challenges in applying off-the-shelf IDS (Snort and Bro) on mesh devices and proposes a lightweight (i.e., customized) IDS for WMN. The proposed IDS requires less memory and decreases the packet drop rate, when compared to off-the-shelf IDS. These achievements, however, are at the price of detecting fewer types of network attacks (smaller detection coverage), since most IDS functions are not implemented. *Cooperative* solutions (e.g., hierarchical [8] or group-based [9] cooperation) distribute IDS functions to multiple cooperative nodes, in order to achieve higher detection rate and lower IDS load. These solutions, however, incur high communication overhead and high latency in attack detection. This is because nodes have to exchange their local observations with other nodes running different IDS functions. Considering the relatively high traffic rates in WMN, caused by mesh clients and external hosts in WMN, the communication overhead of cooperative IDS [9, 10] degrades the network performance and delays intrusion response.

This research is motivated by the fact that neither monitoring nodes nor cooperative IDS techniques can practically solve the intrusion detection problem in WMN. As we will show in Section 2, the fact that WMN are resource constrained poses significant challenges for intrusion detection. Our idea is to use the knowledge a security administrator has about the WMN traffic to distribute IDS functions more efficiently. More precisely, a security administrator, knowing the routing paths of the traffic in the WMN, would employ a traffic-aware framework that optimally places IDS functions on the nodes along the routing paths. The information about the busiest and most frequently used paths in the WMN is obtained from routing algorithms (e.g., OLSR) and network monitoring tools (e.g., tcpdump). Furthermore, it is observed [4] that when deploying WMN for disaster response, the points of interest like physical locations of data sources (e.g., Search & Rescue Robots) and destinations, e.g., Command and Control Center, and consequently the traffic paths are always known.

A related idea for traffic-aware IDS deployments in wired networks was recently proposed [11], where different IDS responsibilities (i.e., different portions of network traffic) are assigned to each node along the traffic paths while ensuring that no node is overloaded. However, that technique cannot be directly

[1] A host inside the mesh is either a client or a local server (e.g., a local FTP server) connected to the mesh routers.

applied to WMN since it assumes that each node performs all IDS functions - infeasible for resource constrained mesh devices. Our proposed solution has no communication overhead, has no detection latency (i.e., it provides real-time intrusion detection, in contrast to cooperative IDS) and it has a higher detection rate, when compared with monitoring node solutions. In our proposed solution, *each node along a routing path, runs a distinct and customized IDS*. This *customized* IDS (technically a subset of IDS functions) allows resource conservation. The combination of *distinct* IDS along the path allows for a complete set of IDS functions to be applied to the entire network traffic. Our main concern in this paper is the reduction of RAM utilization as we will experimentally show that it also improves the CPU utilization in regular traffic rates. More precisely, our paper makes the following contributions: 1) demonstrates that distributing IDS functions along routing paths increases the intrusion detection rate and decreases the average memory load; 2) formulates a novel IDS function distribution problem, called Path Coverage Problem (PCP), with the objective to maximize the detection rate while ensuring that nodes are not overloaded by IDS functions; 3) presents PRIDE, a protocol implemented to solve PCP accurately and fast, based on an Integer Linear Program (ILP); 4) presents results obtained from a real prototype system implementation and an evaluation in a real-world, department-wide, deployed WMN.

2 Motivation and Background

The research presented in this paper is motivated by the challenges we faced when attempted to deploy a common off-the-shelf IDS with a full configuration (i.e., configured to detect the largest set of attacks) on existing WMN router hardware. When loading Snort with its full configuration on a Netgear WNDR3700 router, the router crashes because the RAM is not sufficiently large. In the remaining part of this section we describe the hardware capabilities of our mesh routers, background information on Snort, and experimental results that illustrate how Snort configuration (note: this is equivalent with trading off intrusion detection capabilities) impacts memory load of the router.

The Netgear WNDR3700 router has an Atheros AR7161 processor running at 680MHz, 64MB RAM, 8MB flash memory. It has two wireless cards with Atheros AR9223-bgn and Atheros AR9220-an chipsets, working on 2.4GHz and 5GHz, respectively. The operating system on the router is the most recent release of OpenWrt (i.e., Backfire 10.03.1), a Linux distribution for embedded networking devices, with kernel version 2.6.32.10. We emphasize that our mesh hardware is more powerful (in terms of processing and memory resources) than devices used in some existing real world deployments [2,3]. Although in this research we focus mainly on Netgear WNDR3700 router hardware, later in this section we present our experience and results with more sophisticated and expensive mesh hardware, e.g., Meshlium Xtreme which has a 500MHz CPU, 256MB RAM, 8/16/32GB disk memory and WiFi, Zigbee, and GPRS wireless interfaces.

The router runs Snort, an off-the-shelf intrusion detection system. Snort's detection engine is based on thousands of detection rules (categorized in multiple rule files, corresponding to known network threats) and several preprocessors. All files are listed in "snort.conf", a global configuration file. Upon activating each rule file in "snort.conf" and running Snort, all detection rules present in the rule file are loaded in memory and are used for packet investigation. A full Snort configuration activates all preprocessors and rule files. A customized configuration activates only some preprocessors and rule files (i.e., IDS functions), thus, the network traffic is analyzed by fewer detection functions.

The intrusion detection in Snort is performed by packet-level rule matching. Each packet is preprocessed, following preprocessing directives for extracting possible plain-text content. The preprocessed packet is then inspected by Snort detection rules, to expose whether it is an intrusion attempt or not. Preprocessors parse network packets and provide abstract data for some high-level detection rules in the rule files. It is important to note that a rule file that contains high-level detection rules has *preprocessor dependency*. This dependency means that the rule file cannot be activated (i.e., Snort generates an error message and stops) unless all the preprocessors required by its rules (usually one or two preprocessors) are also activated.

To understand how different Snort configurations impact the memory load on the Netgear WNDR3700 and Meshlium Xtreme, we performed several experiments. Running Snort causes two types of memory loads to the router: 1) *static*, the initial load imposed by packet capturing modules, preprocessors, detection rules, etc. when Snort is loaded; 2) *dynamic*, the variable load imposed by stateful preprocessors (e.g., Stream5) which is a function of the traffic load and some configuration parameters.

We first investigate the static memory load of Snort on the routers when no network traffic is applied. We have observed that a typical memory load on a Netgear WNDR3700 router is ∼30% and on the Meshlium Xtreme router it is ∼60%. This accounts for OS firmware and various services (OLSR, DHCP, etc.). Without preprocessors or rule files active, loading Snort on Netgear WNDR3700 increases memory load to 43% ("Snort(S)" in Figure 1). Memory load increases to 46% if preprocessor Stream5 is activated ("S+str5" in Figure 1), and to 48% if preprocessors "http-inspect", "smtp" and "ftp-telnet" are also activated ("S+4Pre" in Figure 1).

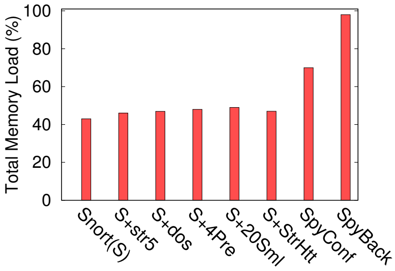


Fig. 1. Effect of Snort configuration on the memory load.

The memory load of a rule file is a function of the number of detection rules in it and the pattern matching algorithm Snort uses (e.g., Aho-Corasick). For example, using "ac-bnfa-nq" search method, "dos.rules" which has 20 detection rules and requires the Stream5 preprocessor, increases memory load to 47% ("S+dos" in Figure 1). A very large file such as "spyware-put" ("SpyConf" in

Figure 1) which contains ~1,000 rule files increases the RAM load to 70%. The memory load caused by activating a set of rule files also depends on their sizes. For example, activating 20 small rule files (i.e., 10 rules per file on average) and the Stream5 preprocessor (which the rules require) increases memory load to 49%. Activating two large rule files, "spyware-put.rules" and "backdoor.rules" ("SpyBack" in Figure 1) increases memory load to 98%. We have experimentally verified that adding a few small rule files on top of "spyware-put.rules" and "backdoor.rules" causes the router to crash. *We have observed a similarly overloaded operation for the Meshlium Xtreme router, where a full configuration Snort increases the memory load to 98.5%, leaving almost no room for processes/services beyond stock deployment.* We also emphasize here the rapid increase in the number of Snort rule files (i.e., currently about 70 files) and their sizes as functions of the number of threats. Some rules might not be needed in a particular setting, but conversely, that setting might require many more rules of some other kind (e.g., custom signatures for suspicious or blacklisted domains, which can increase significantly).

Dynamic memory load, imposed by Stream5 when tracking traffic sessions, is the other considerable type of Snort memory load since almost all rule files require this preprocessor. Two configuration parameters of Stream5, "max_tcp" and "memcap", specify the maximum simultaneous TCP sessions it tracks (similarly, "max_udp", "max_icmp", and "max_ip") and the maximum buffer size for TCP packet storage, respectively. We have experimentally observed that the value of "max_tcp" affects both dynamic and static memory loads. When using the Snort version available on the OpenWrt development tree, the default configuration has max_tcp=8192. Choosing max_tcp=100,000, imposes ~10% more static load than default "S+Str5" to the routers. Moreover, this value allows more simultaneous TCP sessions to be inspected which also imposes larger dynamic memory load and may cause the router to crash at high traffic rates (note: we observed that for max_tcp≥150,000 the router crashes if a simple HTTP request is sent using the Linux "wget" tool). Throughout this paper, we use the default setting, i.e., max_tcp=8192, and consider the maximum dynamic load this setting imposes on the router. Hence, the total memory load of Stream5 is assumed to be its static load plus its maximum allowable dynamic load. We note that although hardware improves, the fundamental challenge for a resource-limited node to handle *ever-increasing network traffic* still remains.

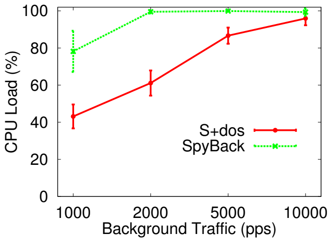


Fig. 2. Effect of Snort configuration on the CPU load.

In addition to RAM, processing power (CPU) is also limited on current mesh hardware. Consequently, investigating the impact of Snort IDS on this limited resource might seem worthwhile. Experimentally we have found that network traffic, actually, has a much larger influence on CPU utilization than executing Snort IDS functions. Our experimental results are depicted in Figure 2 where we enabled "tcp_track" and

"icmp_track" in Stream5 and used "hping3"
to generate TCP and ICMP traffic. As shown, for an extremely high traffic rate, both lightweight and heavy Snort configurations impose more than 95% CPU utilization. Similar with our result, it was shown [7] that even a lightweight IDS exhausted the CPU when traffic rate was extremely high. However, as shown in Figure 2, "S+dos", a lightweight IDS configuration, imposes less processing load than "SpyBack", a heavyweight IDS configuration, when the traffic rate is not high. *Consequently, we aim at reducing the memory utilization as we have experimentally observed that it also improves the CPU utilization in regular traffic rates (as shown in Figure 2).*

3 System and Security models

The system we are considering in this paper is as specified by the IEEE 802.11s WLAN Mesh Standard [1]. The system consists of: i) *mesh access points (AP)* connecting mesh clients (from now on we will refer to them as "clients") to the mesh network; ii) a *wireless mesh backbone*; and iii) a *gateway*, connecting the mesh network to the Internet. The network traffic is either *external*, i.e., between clients and external hosts (external to the mesh), or *internal*, i.e., between two hosts inside the mesh network. Our system also requires the presence of a base station – a computer which periodically and securely collects, via a middleware, information about mesh nodes: *processing/memory* loads, traffic information, etc. Based on these information, the base station assigns IDS functions to nodes.

The IDS we are considering in this paper is Snort. We chose Snort because it is a mainstream off-the-shelf IDS that consumes less resources than other IDS, e.g., Bro (as it was shown recently [7]). Moreover, Snort is readily available for our mesh hardware, as part of the OpenWrt development tree. To the best of our knowledge, there is no port of Bro to the mesh hardware we have available. Assigning a Snort IDS function to a node is equivalent to activating a rule file in the Snort configuration file on that node. Activating a rule file imposes a specific amount of memory load to the device, thus, a limited number of rule files can be activated when running Snort on the device. We use the default search method of Snort, i.e., "ac-bnfa-nq", as we experimentally observed that it consumes the minimum memory among all *low memory* search methods, e.g., "lowmem."

We consider multi-hop attacks where the attacker and the target are connected to the mesh network at different APs. Thus, the attack traffic (malicious packet(s)) is routed across multiple nodes. The attacker can be either *insider* or *outsider*. An insider attacker is a client, connected to a mesh AP, running attacks against a target (a router or host) several hops away. An outsider attacker is an external host attacking a router or a host in the mesh network.

4 Problem Formulation

In this section, we formulate the optimal distribution of IDS functions as an optimization problem and propose a method to solve it. We use Figure 3 to support our formulation. Although Snort is our target IDS (and present a formulation that uses Snort terminology), we believe that other IDS (e.g., Bro) can be analyzed similarly, if their internals and functionality are publicly available.

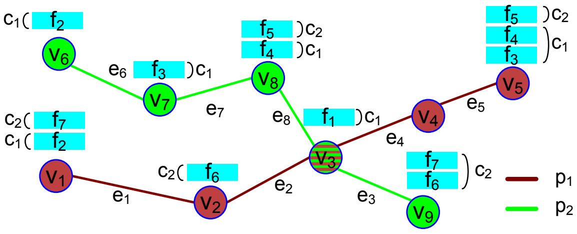


Fig. 3. An example graph for a WMN, consisting of 9 nodes, 8 links, and two paths (p_1 and p_2). The nodes run different configurations of Snort, e.g., node v_5 runs Snort functions f_3, f_4 and f_5, which require preprocessors c_1 and c_2.

We denote the number of nodes and number of links in the wireless mesh network by N and Q, respectively. Considering the information collected from the nodes, we denote the number of nodes and links actively contributing in traffic routing by n ($n \leq N$) and q ($q \leq Q$), respectively. Thus, we model the wireless mesh network (i.e., after removing *idle* nodes/links) as a reduced graph $G = \{V, E\}$, where V is the set of nodes $\{v_1, v_2, \cdots, v_n\}$, and E is the set of links $\{e_1, e_2, \cdots, e_q\}$. An example of a reduced graph, in Figure 3, $V = \{v_1, v_2, ..., v_9\}$ and $E = \{e_1, e_2, ..., e_8\}$.

We denote the set of routing paths for the network traffic by $P = \{p_1, p_2, \cdots, p_l\}$, where $P_i = \{v_j \,|\, v_j \text{ is located in } p_i\}$ and $P_i \subseteq V$. In Figure 3 two paths are present: p_1 and p_2. Additionally, we denote by matrix $\mathbb{T}_{l \times n}$ the mapping between nodes and paths, i.e., $t_{ij} = 1$ iff node j is located on path i. For the example shown in Figure 3, the matrix $\mathbb{T}$ is as follows:

$$\mathbb{T} = \begin{bmatrix} 1\,1\,1\,1\,1\,0\,0\,0\,0 \\ 0\,0\,1\,0\,0\,1\,1\,1\,1 \end{bmatrix}.$$

We denote the set of IDS functions by $\mathcal{F} = \{f_k \,|\, f_k \text{ is a set of detection rules}\}$ with size K (i.e., $|\mathcal{F}| = K$). We denote the set of IDS preprocessors by $\mathcal{C} = \{c_r \,|\, \exists\, f_k \in \mathcal{F} \text{ that requires } c_r\}$ of size R (i.e., $|\mathcal{C}| = R$). For the example in Figure 3, $\mathcal{F} = \{f_1, f_2, ..., f_7\}$ and $\mathcal{C} = \{c_1, c_2\}$. The dependency between IDS functions and preprocessors is stored in matrix $\mathbb{D}_{K \times R}$ where $d_{kr} = 1$ means that activation of function f_k requires the activation of preprocessor c_r.

Let $w : \{\mathcal{F}, \mathcal{C}\} \longrightarrow [0, 1]$ be a cost function that assigns memory load w_k^f and w_r^c to IDS function f_k and IDS preprocessor c_r, respectively. Consequently, vectors $W^f = [w_1^f, w_2^f, \cdots, w_K^f]$ and $W^c = [w_1^c, w_2^c, \cdots, w_R^c]$ represent memory loads for the IDS functions in $\mathcal{F}$ and for the IDS preprocessors in $\mathcal{C}$, respectively (we remark that $w_{Stream5}^c$ in Snort is the summation of its static load and its maximum dynamic load). We denote by $B = [b_1, b_2, ..., b_n]$ the base memory load (i.e., without IDS functions loaded) of all nodes. Finally, we use vector $\Lambda = [\lambda_1, \lambda_2, \cdots, \lambda_n]$ (*i.e., Memory Threshold*) to represent the maximum allowable memory load after IDS functions are loaded. Memory threshold is an important parameter. It is typically set by a network administrator based on the number of active services in the mesh network and the memory space they require.

Definition 1 *An **IDS Function Distribution**, $A = \{(v_j, \mathcal{F}_j, \mathcal{C}_j) |\, v_j \in V,\ \mathcal{F}_j \subseteq \mathcal{F},$ and $\mathcal{C}_j \subseteq \mathcal{C}\}$, is a placement of IDS functions in the network, such that node v_j only executes IDS functions $\mathcal{F}_j$ and their corresponding preprocessors $\mathcal{C}_j$.*

For example, the *IDS Function Distribution* in Figure 3 is:

$$A = \{(v_1, \{f_2, f_7\}, \{c_1, c_2\}), (v_2, \{f_6\}, \{c_2\}), (v_9, \{f_6, f_7\}, \{c_2\})\}.$$

We represent an *IDS Function Distribution* by matrices $\mathbb{X}_{n\times K}$ and $\mathbb{Z}_{n\times R}$, corresponding to IDS functions and preprocessors active on each node, respectively. For $\mathbb{X}$, $x_{jk} = 1$ iff IDS function f_k is activated on node v_j. For $\mathbb{Z}$, $z_{jr} = 1$ iff preprocessor c_r is activated on node v_j. Matrix $\mathbb{Z}$ for the network in Figure 3 is (we omit matrix $\mathbb{X}$ due to space constraints):

$$\mathbb{Z}^T = \begin{bmatrix} 1\,0\,1\,0\,1\,1\,1\,1\,0 \\ 1\,1\,0\,0\,1\,0\,0\,1\,1 \end{bmatrix}.$$

Considering the above mathematical formalism, the dependencies between IDS functions and preprocessors can now be represented more compactly as:

$$z_{jr} = \begin{cases} 1 & if \quad (\mathbb{X}\cdot\mathbb{D})_{jr} \geq 1 \\ 0 & if \quad (\mathbb{X}\cdot\mathbb{D})_{jr} = 0 \end{cases} \tag{1}$$

Equation 1 indicates that preprocessor c_r must be activated on node v_j if there exists at least an IDS function f_k requiring c_r, assigned to it. It is important to note that $z_{jr} = min\{1\,,\, \Sigma_{k=1}^{K} x_{jk} d_{kr}\}$ and $z_{jr} \in \{0, 1\}$.

After the *IDS Function Distribution*, the total memory load for node v_j becomes $L_j = b_j + \Sigma_{c_r \in \mathcal{C}_j} w_r^c + \Sigma_{f_k \in \mathcal{F}_j} w_k^f$, where $w_r^c \in W^c$ and $w_k^f \in W^f$. It is important to mention that an *IDS Function Distribution* in which $L_j > \lambda_j$, i.e., the load L_j is greater than threshold λ_j, for any node v_j, is deemed infeasible.

From a network security administrator point of view, we aim for an *IDS Function Distribution* where all IDS functions are activated on each path. This means that the entire network traffic will be investigated by all IDS functions (albeit at different times), eliminating the possibility of false negatives.

Definition 2 *For a given path p_i and its corresponding set of nodes P_i, **Coverage Ratio (CR)** is defined as $CR_i = |U_i|/K$, where $U_i = \bigcup_{v_j \in P_i} \mathcal{F}_j$ is the set of IDS functions assigned to nodes along the path. Path p_i is called **covered** if $CR_i = 1$ $(U_i = \mathcal{F})$, i.e., for $\forall f_k \in \mathcal{F}$, $\exists\, v_j$ assigned by $\mathcal{F}_j$ such that $f_k \in \mathcal{F}_j$.*

Considering the effect of *IDS Function Distribution* on the memory load of each node and the desired distribution of IDS functions to the nodes, in order to achieve higher intrusion detection rate, we define Path Coverage Problem (PCP) as follows:

Definition 3 ***Path Coverage Problem (PCP)***
Given $G = \{V, E\}$, a set of paths P in WMN, the dependency matrix $\mathbb{D}$, and vectors W^f and W^c, find a distribution $A = \{(v_j, \mathcal{F}_j, C_j) |\, v_j \in V and\ \mathcal{F}_j \subseteq \mathcal{F} and\ C_j \subseteq C\}$, such that $\frac{1}{l}\sum_{p_i \in P} CR_i$ is maximized and $L_j \leq \lambda_j\ \forall v_j \in V$.

PCP is an optimization problem which has the objective of maximizing the average coverage ratio while guaranteeing that memory loads on nodes are below a memory threshold. Although a lower memory threshold λ_j allows more additional processes to execute on node v_j, it makes solving PCP more difficult.

$$\begin{aligned}
&\mathit{Max.} && \frac{1}{l}(\mathbf{1}^T \cdot \mathbb{T})(\mathbb{X} \cdot \mathbf{1}) && && (2)\\
&\mathit{s.t.}: && B^T + \mathbb{Z} \cdot W^{c^T} + \mathbb{X} \cdot W^{f^T} \leq \quad \Lambda^T && && (3)\\
& && (\mathbb{T} \cdot \mathbb{X})_{ik} \leq 1 && , \forall i, k && (4)\\
& && z_{jr} \geq \frac{(\mathbb{X} \cdot \mathbb{D})_{jr}}{K} && , \forall j, r && (5)\\
& && z_{jr} \leq (\mathbb{X} \cdot \mathbb{D})_{jr} && , \forall j, r && (6)\\
& && x_{jk}, z_{jr} \in \{0, 1\} && , \forall j, k, r && (7)
\end{aligned}$$

We formulate PCP as an Integer Linear Program (ILP) that can be solved by an ILP solver. The objective function is maximizing the average coverage ratio of all paths. Additionally, preprocessor dependency and memory threshold are considered as ILP constraints. To better understand the mathematical formulation of the objective function, one can expand the objective function as $\frac{1}{l}\Sigma_{i=1}^{l}\Sigma_{j=1}^{n}\Sigma_{k=1}^{K}t_{ij}x_{jk}$ where $t_{ij} = 1$ means node v_j is located on path p_i and $x_{jk} = 1$ means node v_j is assigned by function f_k. In other words, the average CR has to be maximized. Constraint 3 limits the memory load on every node v_j, i.e., $\Sigma_{r=1}^{R}z_{jr}w_r^c + \Sigma_{k=1}^{K}x_{jk}w_k^f$, to be less than its memory threshold λ_j. Most importantly, (*to ensure that we can formulate PCP as a linear program*), this constraint computes the total memory load as the sum of individual memory loads of preprocessors and rule files. Obviously, one needs to investigate if this linearity assumption always holds (we will discuss this in the next section). Constraint 4 ensures that only one copy of each function is assigned to the nodes along each path. Constraints 5 and 6 ensure that if an IDS function is assigned to a node, its required preprocessors are also assigned to the node. As presented in Equation 1, $z_{jr} = 1$ if at least one of the IDS functions assigned to node v_j requires preprocessor c_r, otherwise $z_{jr} = 0$. The maximum number of functions that require a specific preprocessor is at most K. Hence, Constraint 5 ensures that $0 < z_{jr} \leq 1$ if there is a function assigned to node v_j that requires preprocessor c_r. On the other hand, if none of the functions assigned to node v_j requires preprocessor c_r, then Constraint 6 enforces z_{jr} to be zero. Taking into account Constraint 7, i.e., z_{jr} has to be either 0 or 1, Constraint 5 enforces $z_{jr} = 1$ if preprocessor r is required on node j, otherwise, Constraint 6 enforces $z_{jr} = 0$.

5 PRIDE: Challenges and Solutions

Considering the aforementioned ILP formulation for PCP, we investigated two major challenges that impact the accuracy and time complexity of a solution. First, we experimentally observed that the total memory load of multiple Snort rule files is generally linear (i.e., it is equal to the sum of their individual memory loads), but not always (e.g., for some small rule files and certain rule types). This influences the accuracy of our proposed model for calculating the total memory load on each node (i.e., Challenge 1). Next, one can observe that the complexity of ILP depends on the number of paths in the network, the path lengths, the number of IDS functions, the number of preprocessors, and the memory threshold. For example, considering the number of Snort preprocessors (i.e., more than 20) and the number of Snort rule files (i.e., more than 60), *for single path* p_i, the number of ILP constraints grows to more than $1400 \times |P_i|$, where $|P_i|$ is the path length. Additionally, a lower memory threshold λ_j increases the number of infeasible solutions, thus requiring more iterations for the ILP solver. Hence, the ILP performance degrades as network size increases or memory threshold decreases (i.e., Challenge 2). In this section, we investigate the aforementioned challenges and propose techniques to overcome them. Finally, we present PRIDE protocol that distributes IDS functions to the nodes accurately and fast (i.e., practical).

Experimentally, we observed that when activating multiple *small* rule files (i.e., containing at most 50 detection rules), Snort memory load is much less than the sum of individual memory loads. However, we observed that when multiple *large* rule files (i.e., containing more than 250 detection rules) were activated, the memory load is closer to the sum of the rule file's individual memory loads. When a rule file is activated, depending on: 1) the number of detection rules it has; 2) the preprocessors it activates (if already not activated); and 3) the Snort search method, a different amount of memory load will be imposed to the node. In order to show how the aforementioned three factors impact our assumption about memory load linearity (i.e., constraint 3), we performed extensive experiments (omitted here due to space constraints) on the Snort memory consumption modeling in the absence of preprocessors. As the result, we observed *a linear behavior when adding blocks of 250 rules to the set of active rules irrespective of rule order and search method.* We use this finding to address the non-linearity of memory load for the variable-size rule files (i.e., Challenge 1) in the following subsections.

5.1 Rule Files Modularization

To reduce the complexity of the problem the ILP solver faces (i.e., Challenge 2), we propose to reduce the number of individual preprocessors and IDS functions, which would result in a decrease in the number of constraints in ILP. Our proposal is to group multiple IDS functions together and consider them as a single function. *From here on, we refer to each group of rule files as a "detecting module" and use the term "group" for a group of preprocessors.* If a detecting module is assigned to a node, all rule files in that module will be activated. We experimentally observed that grouping rule files not only reduces the problem complexity (Challenge 2), but also decreases the variance in memory load estimation (Challenge 1). When several small rule files are grouped in a single detecting module, it acts as a larger rule file (same as a block of 250 rules), and the estimated memory load is more accurate. In addition, considering the *preprocessor dependency* mentioned in Section 4, an efficient rule file grouping reduces the number of preprocessor dependencies. For example, if two rule files require the same preprocessor(s), they can be grouped in the same detecting module. Similarly, multiple preprocessors required for the same rule files, can be grouped together. Hence, when activating a new *detecting module*, the load imposed by rules' data structure dominates the load imposed by the new activated preprocessor (that can be ignored). This is very similar to the behavior observed in memory consumption modeling experiments in the absence of preprocessors.

Grouping rule files together, however, has a disadvantage when the memory threshold set by the system administrator is very low. For low memory thresholds, we cannot assign larger modules to nodes, which results in low coverage/detection ratio. Consequently, despite the positive aspects of grouping small rule files together, memory threshold forces us to avoid large detecting modules. Unfortunately, there already exist large detecting modules. For example, the memory space required by the "backdoor" rule file is twice the memory space required by a detecting module with 25 small rule files. This illustrates the need to also split extremely large rule files into some smaller ones (i.e., creating several detecting modules out of a large rule file).

We thus define "modularization" as the procedure that, for a given set of IDS functions (e.g., Snort rule files), i) *groups* small IDS functions together in order to reduce the problem complexity and load estimation error, and ii) *splits* large IDS functions into several smaller functions so that they can be activated with low memory thresholds.

Rule File Splitting: When splitting a rule file, we consider the dependency between detection rules and the dependency between preprocessors and detection rules. This is to ensure that two dependent rules along with all of their essential preprocessing directives are included in the same split rule file. In order to split a rule file into several detecting modules, we first pre-parse each detection rule and specify its preprocessing dependency in advance (e.g., Stream5 preprocessor for HTTP-relevant rule files). We summarize all these preprocessing dependencies before splitting the rule files. In addition, rule dependency is expressed by the options' keywords, e.g., "flowbits." To meet the rule dependency requirements, we parse each detection rule and specify whether the rule contains such keywords or not, if it does, it must be relevant. For example, the "flowbits" options can help us maintain the stateful check in a set of Snort detection rules. When some keys are set by "flowbits" in a detection rule, every other detection rule which does not set the "flowbits," is dependent on that detection rule. Thus, using these two types of dependency, we split large rule files properly.

In addition, rule dependency is expressed by the options' keywords, e.g., "flowbits." To meet the rule dependency requirements, we parse each detection rule and specify whether the rule contains such keywords or not, if it does, it must be relevant. For example, the "flowbits" options can help us maintain the stateful check in a set of Snort detection rules. When some keys are set by "flowbits" in a detection rule, every other detection rule which does not set the "flowbits," is dependent on that detection rule. Similarly, the keyword "rev:VALUE" in a detection rule, that identifies revisions of Snort rules, denotes that it is related to a detection rule whose "sid" is "VALUE." Thus, using these two types of dependency, we split large rule files properly.

Proposed Modularizations: We propose three modularizations with different numbers of detecting modules and different sizes. We then compare the execution time of the solver, i.e., Matlab ILP solver, for each modularization.

In the first modularization, the entire set of Snort rule files is classified into 23 detecting modules where 6 different groups of preprocessors are required. The average memory load of the 23 detecting modules is 3.98% and the standard deviation is 1.68%. The second modularization consists of 12 detecting modules of average memory load 6.76% and standard deviation 2.31%, while the third modularization has only 6 detecting modules of average memory load 15.06% and standard deviation 1.88%. The second and the third modularizations require three groups of preprocessors.

Figure 4 shows the execution time of the ILP solver when solving the problem for different lengths of a single path. As depicted, 12-module and 6-module configurations are much faster than 23-module configuration, especially for longer paths (i.e., more complex problems). With these two modularizations, the ILP solver finds the optimal solution in less than 2 sec, which is very fast, thus practical in real deployments. The longer execution time for 6-module configuration, comparing to 12-module configuration, is because of its larger detecting modules that increase the number of infeasible solutions for a given memory threshold (increasing the solver's execution time). We use 6-module and 12-module configurations in our system evaluations.

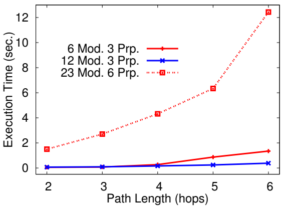


Fig. 4. Effect of modularizations on ILP execution time.

5.2 PRIDE Protocol

Given a modularization chosen for the IDS configuration, PRIDE periodically collects the local information from the nodes, removes *idle* nodes from the network, i.e., those not contributing in the traffic routing, and optimally distributes IDS functions to the nodes along traffic paths. If the reduced graph is disconnected, each graph component is considered as a sub-problem and solved separately. Algorithm 1 presents PRIDE protocol.

Algorithm 1 PRIDE IDS Function Distribution

```
1:  $Data\_Collection(V, E, N, Q)$
2:  $Relaxation(V, E, n, q)$
3:  $Path\_Extract(V, E, P)$
4:  $\mathcal{P} = P$
5:  $g = 0$
6:  while $\exists p_i \in \mathcal{P}$ do
7:      $g++$
8:      $S_g = \{p_i\}$
9:      $\mathcal{P} = \mathcal{P} \backslash \{p_i\}$
10:     while $\exists p_j \in Q$ and
11:         $\bigcup_{p_k \in S_g} (P_j \cap P_k) \neq \varnothing$ do
12:         $S_g = S_g \cup \{p_j\}$
13:         $\mathcal{P} = \mathcal{P} \backslash \{p_j\}$
14:     end while
15: end while
16: for $\forall S_g$ do
17:     $V_g = \{v_j | v_j \in P_i \, and \, p_i \in S_g\}$
18: for $\forall V_g$ do ILP$(V_g, P)$
```

Given the set of nodes, the protocol first collects information from nodes and then produces the reduced sets V and E by removing idle nodes/links. Next, the set of active routing paths P is extracted in Line 3. Given P, the Algorithm creates the set $\mathcal{P}$ of unvisited paths, and then defines variable g as the number of sub-problems. For every unvisited path p_i in set $\mathcal{P}$, the Algorithm first creates a new sub-problem S_g and marks it as a visited path. The Algorithm then searches $\mathcal{P}$ to find any unvisited path p_j which is *connected* (Two paths are *connected* if they are in the same component of the reduced graph) to at least one path in the current S_g. If so, the corresponding path p_j will be added to the current sub-problem S_g and removed from $\mathcal{P}$. When no more paths in $\mathcal{P}$ can be added to the current S_g, the Algorithm increases g and creates a new sub-problem. This process repeats until there is no unvisited path in $\mathcal{P}$. Next, for every sub-problem S_g, the Algorithm creates the corresponding set V_g as the set of nodes located on the paths of component S_g. Finally, the Algorithm runs ILP on the nodes and paths of each sub-problem S_g.

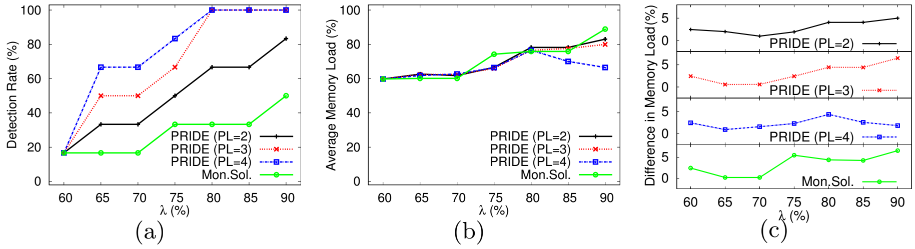
6 System Implementation and Evaluation

In this section, we evaluate the performance of PRIDE in a department-wide mesh network. Our mesh network consists of 10 Netgear WNDR3700 routers deployed in a $50 \times 30m^2$ rectangular area (Note: comparing with other testbeds, DistressNet [4] 8 nodes, SMesh [2] 14 nodes, PRIDE uses an average size testbed.). The routers use OLSR as the routing protocol and provide mesh connections on their 5GHz wireless interfaces and network access for the clients on the 2.4GHz wireless interfaces. PRIDE periodically (i.e., 5 minutes in the current setup) collects nodes/traffic information and runs ILP. This interval can be optimally chosen by administrator in dynamic networks. We use *bintprog* function in Matlab as the ILP solver.

We evaluate the *intrusion detection rate (coverage ratio)* and *average memory load* of nodes. The parameters that we vary are the *Path Length (PL)* and *memory threshold* (λ). The attack traffic we use is based on *Rule 2 Attack* tool, as explained in [12]. In all our experiments, the memory thresholds of all nodes are equal and some of the preprocessors (e.g., perfmonitor) are not used as they are not activated by default or not required by rule files. Since the maximum path length in our mesh network is 4 hops, we consider 2-hop, 3-hop and 4-hop paths. The initial memory load on the routers is $\sim 30\%$ (as caused by DHCP, OLSR, and other services). We vary the Snort memory threshold from 30% to 60% (i.e., $60\% \leq \lambda \leq 90\%$). Since implementing the related traffic-aware solution [11] on the mesh devices is infeasible (the routers crash), we compare PRIDE with monitoring node solutions ([5,6]). We implement a monitoring node solution [5] to which we refer as "MonSol". A monitoring node loads detecting modules up to a given memory threshold based on the default order of rule files in Snort configuration file. If a monitoring node monitors at least one link of a given path, the entire path is considered as monitored.

6.1 Proof-of-Concept Experiment

When assigning IDS functions to multiple nodes on a path, each node can detect only a subset of attacks depending on the detecting modules it executes. As a proof-of-concept experiment, we show that distributing two IDS functions to two nodes generates exactly the same alerts as if both detecting modules were assigned to a single node (e.g., MonSol). For that purpose, we used two routers and one laptop connected wireless to each router (one laptop was the attacker



(a) (b) (c)

Fig. 5. 6-module configuration: effect of λ and PL on a) Detection rate. b) Average estimated memory load. c) The difference between estimated and actual memory load.

and the other was the target). We ran a customized Snort on each router (monitoring the mesh traffic) ensuring that every Snort rule file is activated on at least one of the routers. We then generated two R2A exploits such that their corresponding rule files, e.g., “dos.rules” and “exploit.rules”, were activated on routers 1 and 2, respectively. When running attacks, the Snort on node 1 generated 4 alerts, while the one on node 2 generated 10 alerts (real-time detection, unlike cooperative IDS). We repeated the experiment where only node 1 was running Snort and both rule files were activated on node 1 (many other rule files were deactivated due to memory constraint). In this experiment, node 1 generated exactly the same 14 alerts upon launching the same exploits. Hence, we have shown that PRIDE can distribute IDS functions to nodes along a path such that network packets are inspected by all IDS functions.

6.2 Effects of Memory Threshold and Path Length

Given the network paths in our test-bed mesh network, we evaluate the intrusion detection rate of PRIDE and the average memory load on nodes, using 6-module and 12-module configurations. For each modularization, we change λ and PL as our evaluation parameters to see their effects on PRIDE performance. Given a λ, we show PRIDE can achieve higher detection rate than MonSol.

Figure 5 shows the effect of memory threshold and path length on intrusion detection rate and average memory load on the nodes when using the 6-module configuration. As depicted in Figure 5(a), maximum detection rate for MonSol is 50% which occurs when $\lambda = 90\%$. However, PRIDE can achieve 100% detection rate even in a lower memory threshold (e.g., at $\lambda = 80\%$ for $PL = 4$ and $PL = 3$). This is because more than one node is assigned with IDS functions and packets are inspected by more detecting modules. In this modularization, for a low memory threshold (e.g., $\lambda = 60\%$), only module 3 can be activated on the nodes, and thus, PRIDE cannot achieve a higher detection rate than MonSol. Figure 5(b) depicts the average estimated memory load on the nodes for different memory thresholds. It can be observed that PRIDE usually requires less memory load than MonSol, especially for the longer paths, since the modules are distributed to multiple nodes. We also compare the estimated memory loads and the actual memory loads of the two configurations in all of the experiments, i.e., different memory thresholds and path lengths. Figure 5(c) shows the differ-

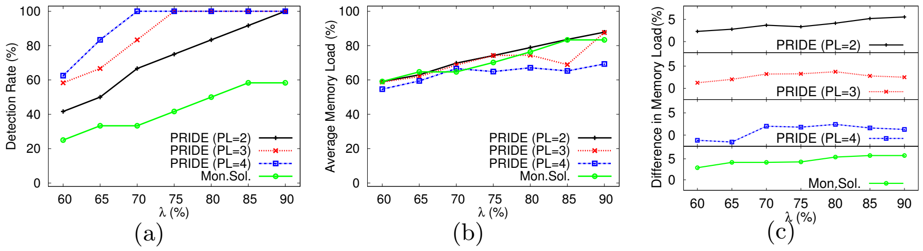


Fig. 6. 12-module configuration: effect of λ and PL on a) Detection rate. b) Average estimated memory load. c)The difference between estimated and actual memory load.

ence between estimated memory load and actual load measured on the routers when using 6-module configuration. One can see that the difference is below ∼5%, thus giving confidence in our ILP formulation and memory consumption modeling. The results for the same evaluations performed on the 12-module configuration are shown in Figure 6. As depicted in Figure 6(a), the intrusion detection rate for the 12-module configuration is higher than the detection rate for the 6-module configuration (for the same memory threshold). This is because the size of the detecting modules in the 12-module configuration is smaller than for the 6-module configuration, which allows more modules to fit in the small free memory spaces. In contrast with the 6-module configuration, where at low memory thresholds the detection rate was similar to MonSol, in the 12-module configuration the detection rate at 60% (a low memory threshold) is higher than for MonSol. This is because more modules are activated on the nodes even at this low memory threshold. The average estimated memory loads for this modularization are shown in Figure 6(b). Similar to the 6-module configuration, it is observed that the 12-module configuration usually impose less memory load than MonSol solution for the longer paths. It is worth mentioning that the estimated values for the 12-module configuration, as shown in 6(c), are closer to the real values than the 6-module configuration because the modules are roughly the same size as 250-rule blocks.

Figure 7 shows the ILP solver execution time for $PL = 3$ and $PL = 4$, and for each modularization. As depicted, the execution time of the algorithm ranges from a few seconds to tens of seconds, thus making it practical for real world deployments. As shown, the lower the memory threshold is, the longer the execution time is. This is because lower memory thresholds increase the number of infeasible solutions and the solver requires more iterations to obtain feasible and optimal solutions. As shown in Figure 7, the execution time increases with the path length as well. As mentioned in Section 5, this is because the number of ILP constraints (i.e., the problem complexity) is a direct function of path length.

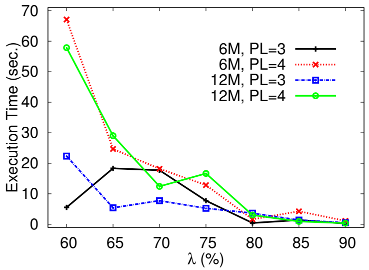


Fig. 7. ILP solver execution time for different problems and parameters.

7 Conclusions

In this paper, we have shown that intrusion detection in WMN requires significant resources, and traditional solutions are not practical for WMN. To address these challenges, we propose a solution for an optimal distribution of IDS functions. We formulate the optimal IDS function distribution as an integer linear program and propose algorithms for solving it accurately and fast. Our solution maximizes intrusion detection rate, while maintaining the memory load below a threshold set by network administrators. We have investigated the performance of our proposed solution in a real-world, department-wide, deployed WMN.

Acknowledgement

This work is based in part on work supported by Naval Surface Warfare Center, Grant No. N00164-11-1-2007.

References

1. G. R. Hiertz, D. Denteneer, S. Max, R. Taori, J. Cardona, L. Berlemann, and B. Walke, "IEEE 802.11s: the WLAN mesh standard," *Wireless Commun.*, 2010.
2. Y. Amir, C. Danilov, R. Musăloiu-Elefteri, and N. Rivera, "The SMesh wireless mesh network," *ACM Transactions on Computer Systems*, September 2008.
3. J. Backens, G. Mweemba, and G. Van Stam, "A rural implementation of a 52 node mixed wireless mesh network in Macha, Zambia," *EInfrastructures and EServices on Developing Countries*, pp. 32 – 39, 2010.
4. H. Chenji, A. Hassanzadeh, M. Won, Y. Li, W. Zhang, X. Yang, R. Stoleru, and G. Zhou, "A wireless sensor, adhoc and delay tolerant network system for disaster response," LENSS-09-02, Tech. Rep., 2011.
5. A. Hassanzadeh, R. Stoleru, and B. Shihada, "Energy efficient monitoring for intrusion detection in battery-powered wireless mesh networks," in *ADHOC-NOW*, 2011.
6. D.-H. Shin, S. Bagchi, and C.-C. Wang, "Distributed online channel assignment toward optimal monitoring in multi-channel wireless networks." in *IEEE INFOCOM*, 2012.
7. F. Hugelshofer, P. Smith, D. Hutchison, and N. J. Race, "OpenLIDS: a lightweight intrusion detection system for wireless mesh networks," in *MobiCom*, 2009.
8. A. Hassanzadeh and R. Stoleru, "Towards optimal monitoring in cooperative ids for resource constrained wireless networks," in *IEEE ICCCN*, 2011.
9. I. Krontiris, Z. Benenson, T. Giannetsos, F. C. Freiling, and T. Dimitriou, "Cooperative intrusion detection in wireless sensor networks," in *Wireless Sensor Networks*. Springer-Verlag, 2009.
10. A. Hassanzadeh and R. Stoleru, "On the optimality of cooperative intrusion detection for resource constrained wireless networks," *Computers & Security*, 2013.
11. V. Sekar, R. Krishnaswamy, A. Gupta, and M. K. Reiter, "Network-wide deployment of intrusion detection and prevention systems," in *ACM CoNEXT*, 2010.
12. A. Hassanzadeh, Z. Xu, R. Stoleru, and G. Gu, "Practical intrusion detection in resource constrained wireless mesh networks," Texas A&M University 2012-7-1, Tech. Rep., 2012.